Homework 4

Due by **11:59pm, Thursday, October 4, 2018**.

Make sure you follow all the homework policies (http://www-student.cse.buffalo.edu/~atri/cse331/fall16/policies/hw-policy.html).

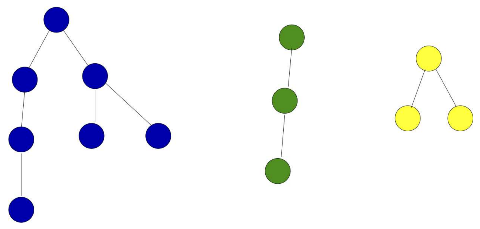
All submissions should be done via Autolab (http://www-student.cse.buffalo.edu/~atri/cse331/fall16/autolab.html).

Sample Problem

The Problem

This problem is just to get you thinking about graphs and get more practice with proofs.

A **forest** with c *components* is a graph that is the union of c disjoint trees. The figure below shows for an example with $c = 3$ and $n = 13$ with the three connected components colored blue, read and yellow).



Note that a tree is a forest with 1 component.

Prove that an n-vertex forest with c components has $n - c$ edges.

Note

Recall we have proved the statement above for $c = 1$ in class.

Click here for the Solution

Submission

You will **NOT** submit this question. This is for you to get into thinking more about proving properties of graphs.

Question 1 (Programming Assignment) [30 points]

</> Note

This assignment can be solved in either Java, Python or C++ (you should pick the language you are most comfortable with). Please make sure to look at the supporting documentation and files for the language of your choosing.

The Problem

In this problem we will explore graphs and problems that are relatable to the 6 degrees of separation problem, which claims that everyone in the world is connected by 6 people or less. We will investigate different networks to see the minimum distance between some starting node, s, and all other nodes in the graph. We will be using real-life data sets to test your code.

We are given a starting node s for some undirected graph G with n nodes. The graph is formatted as an adjacency list, meaning that for each node, u, we can access all of u's neighbors. The goal is to output all n nodes in G along each u's **minimum** distance from s.

Input

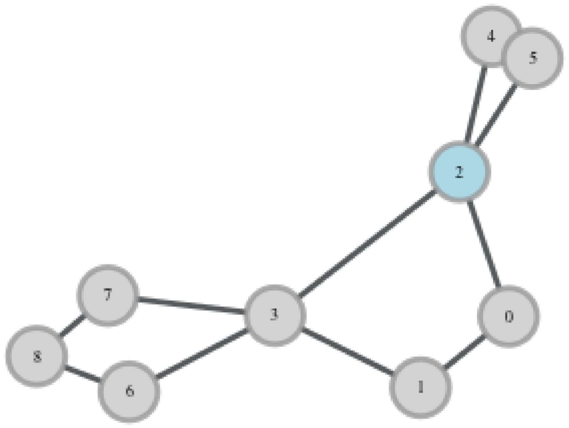
The input file is given with the first line as the starting node and the remainder of the file is an adjacency list for graph G (we assume that the set of vertices is $\{0, 1, \ldots, n-1\}$). The adjacency list assumes that the current node is the index of the line under consideration. For instance line 0 of the input file (not including the starting node) has the list of all nodes adjacent to node 0.

```
s                <- Starting node (some node between u_0 and u_n-1)
u_1 u_4 u_6      <- All nodes that share edges with u_0
u_3 u_u          <- All nodes that share edges with u_1
u_0              <- All nodes that share edges with u_2
.
.
.
u_0 u_4 u_2 u_7  <- All nodes that share edges with u_n-1
```

For example

```
2               <- Starting node
1 2             <- Node 0 shares edges with nodes 1 & 2
0 3             <- Node 1 shares edges with nodes 0 & 3
0 3 4 5         <- Node 2 shares edges with nodes 0, 3, 4 & 5
1 2 6 7         <- Node 3 shares edges with nodes 1, 2, 6 & 7
2               <- Node 4 shares edges with node 2
2               <- Node 5 shares an edge with node 2
3 8             <- Node 6 shares an edge with nodes 3 & 8
3 8             <- Node 7 shares edges with nodes 3 & 8
6 7             <- Node 8 shares edges with nodes 6 & 7
```

If we draw the graph using the above input, it will look like this:



Output

The output is every node (given by the index in the array), along with it's minimum distance from the starting node formatted as your language of choice's array-type data structure (see below for the exact details).

```
[d0 d1 d2... dm]                                     <- Node 0 is distanc
                                                        Node 1 is distanc
                                                        Node 2 is distanc
                                                        .
                                                        .
                                                        .
                                                        Node m is distanc
```

For example, using the example input from above:

```
            [1, 2, 0, 1, 1, 1, 2, 2, 3]                  <- Node 0 is distance 1 from
                                                            Node 1 is distance 2 from
                                                            Node 2 is distance 0 from
                                                            Node 3 is distance 1 from
                                                            Node 4 is distance 1 from
                                                            Node 5 is distance 1 from
                                                            Node 6 is distance 2 from
                                                            Node 7 is distance 2 from
                                                            Node 8 is distance 3 from
```

</> Note

There is **no** guarantee that the graph has only one connected component. In the situation where a node can not be reached from the starting node, the distance returned by your algorithm for that node should be -1.

Hint

The best possible algorithm for this problems that we are aware of runs in time $O(m+n)$ where m in the total number of edges, and n is the total number of nodes.

! Note

Both the input and output parsers in each of the three languages are already written for you. Note that you have to work with the input data structures provided (which will come pre-loaded with the data from an input file).

! Addition is the only change you should make

Irrespective of what language you use, you will have to submit just one file. That file will come pre-populated with some stuff in it. You should **not** change any of those things because if you do you might break what the grader expects and end up with a zero on the question. You should of course add stuff to it (including helper functions and data structures as you see fit).

Java Python C++

Download Java Skeleton Code (HW4Java.zip)

Directory Structure

```
├── src
│   ├── ub
│       ├── cse
│           ├── algo
│               ├── Driver.java
│               ├── Graph.java
│               ├── HW4Utility.java
│               ├── Solution.java
├── testcases/
│   ├── input1.txt
│   ├── input2.txt
│   ├── input5.txt
│   ├── output1.txt
│   ├── output2.txt
│   └── output5.txt
```

You are given four coding files: `Driver.java` , `Graph.java` , `HW4Utility.java` , and `Solution.java` . `Driver.java` takes the input file, parses it with a new instance of `HW4Utility` and creates an instance of the class `Solution` and calls the `outputDistances()` method on it. It then prints the distances (which, if your code is correct, will be the shortest distace from the start node). You only need to update the `Solution.java` file.

On top of that, you may write your own helper methods and data structures in it.

The testcases folder has 3 input files and their corresponding output files for your reference. We will use these three input files (and seven others) in our autograding.

Method you need to write:

```
public int[] outputDistances() {
    return null;
}
```

The `Solution` class has 2 instance variables:

- `startNode` which is of type int and stores the starting node id.
- `graph` which is of type `HashMap<Integer, ArrayList<Integer>>` where the key is the node id and the value is an arraylist of adjacent nodes.

The output is an array of `int` s. In particular, the location `i` (for $0 \leq i < n$) contains the distance of node `i` from `startNode` . Finally, note that the starting node is distance 0 from itself.

Compiling and executing from command line:

Assuming you're in the same directory level as `src` . Run `javac src/ub/cse/algo/*.java` to compile.

To execute your code on `input1.txt` , run `java -cp "src" ub.cse.algo.Driver testcases/input1.txt` . The output array will be printed to `stdout` .

Submission

You only need to submit `Solution.java` to Autolab.

Grading Guidelines

We will follow the usual grading guidelines for programming questions (../../policies/hw-policy.html#grading).

! For those of you who are feeling a little ambitious

For the top 3 submissions in the scoreboard in both Java and Python and for the top 2 submissions in the scoreboard in C++, we are offering 3 bonus points. But be warned! You should not be spending too much time on this. We rather you work on Questions 2 and 3 below.

Question 2 (Seating ADFs) [45 points]

The Problem

There is a group of p Ava DuVernay (http://www.avaduvernay.com/) fans (or ADFs for short) who have booked an entire movie theater for a screening of A Wrinkle in Time (http://www.imdb.com/title/tt1620680/). Out of the $p(p-1)/2$ possible pairs, f pairs of ADFs are friends. The movie theater is called the Square Theater and has p rows with each row having exactly p chairs in them. (If it helps, you can assume that the p^2 chairs are arranged in a "grid"/"matrix" form where each of the p rows has p chairs each and each of the p columns has p chairs each.)

The ADF organization committee has come to you to help them efficiently solve the following seating problem for them. Informally, you want to seat the ADFs so that friends can talk to each other (and "distant" friends remain distant). Next, we make this informal description more formal.

A *seating* (as you might expect) is an assignment of the p ADFs to the p^2 chairs in the theater so that each ADF gets his/her own chair.

A seating is called `admissible` if

1. for every pair of friends (b_1, b_2), they can `talk` to each other during the movies
2. some ADF is assigned a seat in the first row and
3. the seating satisfies the `distance compatible` property.

The ADFs have been to the Square theater before so they have figured out that two people can `talk` to each other if and only if they are either

- seated in the same row or
- are seated in the rows next to them (i.e. either to the row directly in front or the row directly behind. The first and the last row of course only have one row next to them).

A seating has the `distance compatible` property if and only if the following holds. For any b_1 who is assigned a seat in the first row and any ADF b_2, such that b_1 and b_2 have a *friendship distance* of d (assuming d is defined) are seated d or more rows apart. (The friendship distance is the natural definition. Let a *friendship path* between b_1 and b_2 be the sequence of ADFs $f_0 = b_1, f_1, \dots, f_{k-1}, f_k = b_2$ (for some $k \geq 0$) such that for every $0 \leq i \leq k-1$, (f_i, f_{i+1}) are friends-- the length of such a path is k. The friendship distance between b_1 and b_2 is the shortest length of any friendship path between them. So if b_1 is b_2's friend, they have a friendship distance of 1 and if b_2 is a friend of a friend, they have a distance of 2 and so on. The friendship distance is not defined if there is no friendship path between b_1 and b_2.

Note

Note that the above property is **not** defined for **all** pairs of ADFs b_1 and b_2. (Indeed the problem you will be asked to solve is *impossible* if the `distance compatible` property has to hold for all pairs.) Note that the above is defined for only those b_1 who are seated in the first row (and every possible b_2).

As an example consider the case of $p = 3$ and $f = 2$ as follows. The ADFs are A, B, C and the (A, B) and (B, C) are the pairs of friends. Then an admissible seating is to assign the "left-most" seat on first row to A, the left-most seat in second row to B and the left-most seat on the third row to C. This following is also an admissible seating: B is assigned (any seat) in the first row while A and C are assigned any two seats in the second row.

Your final task is to design an efficient algorithm that computes an admissible seating, given as input the set of p ADFs and the set of f pairs of ADFs who are friends. Here are the two parts:

- Part `(a)` : Formalize the problem above in terms of graphs: i.e. write down
    i. How you would represent the input as a graph G;
    ii. How you would define a seating and
    iii. Define what it means for a seating to be admissible in terms of properties of graph G.

  Note

  This should not be too tricky. Also feel free to skip this part if you want to answer part `(b)` directly.

- Part `(b)` : Using part 1 or otherwise design an efficient algorithm to compute an admissible seating. (Recall that your algorithm should work for any set of p people and any possible set of f friendships.) You should prove the correctness of your algorithm. You also need justify the running time of your algorithm.

  Hint

  Think how you can use some of the graph exploration algorithms we have seen in class.

  Common Mistake

  A common mistake in this (and related) problem is to think that both BFS and DFS are completely interchangeable. There are concrete cases where they are not (../../../support/dfs-bfs/index.html).

Walkthrough video

-

Submission

You need to submit **one PDF** file to Autolab. We recommend that you typeset your solution but we will accept scans of handwritten solution-- you have to make sure that the scan is legible.

! PDF only please

Autolab might not be able to display files in formats other than PDF (e.g. Word cannot be displayed). **If Autolab cannot display your file, then you will get a zero (0) on the entire question. Note that Autolab will NOT give an error message if you submit non-PDF file, so it is YOUR responsibility to make sure you submit in the correct format.** Also the file size has to be **at most 3MB**.

Grading Guidelines

We will follow the usual grading guidelines for non-programming questions (../../policies/hw-policy.html#grading). Here is a high level grading rubric specific to part `(a)` of this problem:

1. `Formalizing the problem` : `10` points.

and here is the high level grading rubric for part `(b)` :
We will follow the usual grading guidelines for non-programming questions (../../policies/hw-policy.html#grading). Here is a high level grading rubric specific to this problem:

1. `Algorithm idea` : `9` points.
2. `Algorithm details` : `9` points.
3. `Proof of correctness idea` : `7` points.
4. `Proof details` : `6` points.
5. `Runtime analysis` : `4` points. Note that you only need a Big-Oh analysis.
6. `Note:` If your solution solves part `(b)` directly, then `5` points get added to algorithm idea and `5` points get added to proof idea.

! Note

If you do not have separated out algorithm idea, algorithm details, proof idea proof details and runtime analysis for part `(b)` , you will get a zero(0) irrespective of the technical correctness of your solution..

Templates

Download LaTeX template. (latex-template.zip)

Download Microsoft Word template. (word-template.zip)

! Note

Your must explicitly list your sources and collaborators when you upload your submission to Autolab. Note that there are only five allowed sources (../../policies/hw-policy.html). If you have used a source that is not allowed, please do **not** submit your homework. If you did not consult any source or did not collaborate with anyone just say `None` .

Question 3 (Average vs. Max) [25 points]

The Problem

Let $G = (V, E)$ be a *connected* graph. Recall that the distance between two nodes $u, w \in V$, which we will denote by $d(u, w)$ is the length of the shortest path connecting u and w. In this problem we will consider how far apart the maximum and average of all these distances can be.

More formally, define the maximum distance between any two nodes, also called the *diameter* is defined as

$$diam(G) = \max_{u,w \in V} d(u, w).$$

Similarly define the *average distance* to be

$$avgd(G) = \frac{\sum_{u \neq w \in V} d(u, w)}{\binom{|V|}{2}}.$$

Recall that a complete binary tree (https://en.wikipedia.org/wiki/Binary_tree) is a tree such that all interior nodes have exactly two children *and* all leaves are at the same level. (Note that the Wikipedia page calls such a tree a *perfect binary tree*, though we will stick with the complete tree nomenclature.)

Here are the two parts of the problem:

- Part `(a)` : Argue that a complete binary tree T with depth d has $diam(T) = 2d$.
- Part `(b)` : Argue whether the following claim is true or false. There exists a constant $c \geq 1$ such that for every complete binary tree T, it is the case that

$$\frac{diam(T)}{avgd(T)} \le c.$$

In the above we will use the convention that $\frac{0}{0}$ is actually 1. (This is to get rid of annoying case when T has exactly one node.)

Note

If you claim the statement is false then you need to provide a counter-example and you need to argue why your counter-example disproves the above statement. If you claim the statement is true, then you need to prove it.

Hint

The claim is false for general graphs G. What this problem is asking you to decide is if the statement still hold if we only look at the restricted class of complete binary trees.

Walkthrough video

Submission

You need to submit **one PDF** file to Autolab. We recommend that you typeset your solution but we will accept scans of handwritten solution-- you have to make sure that the scan is legible.

! PDF only please

Autolab might not be able to display files in formats other than PDF (e.g. Word cannot be displayed). **If Autolab cannot display your file, then you will get a zero (0) on the entire question. Note that Autolab will NOT give an error message if you submit non-PDF file, so it is YOUR responsibility to make sure you submit in the correct format.** Also the file size has to be **at most 3MB**.

Grading Guidelines

We will follow the usual grading guidelines for non-programming questions (../../policies/hw-policy.html#grading). Here is a high level grading rubric specific to part `(a)` of this problem:

1. `Proof Idea` : `10` points.

and here is the high level grading rubric for part `(b)` :

1. `Proof idea` : `8` points for the idea behind your counter-example or your argument for why the statement is true.
2. `Proof details` : `7` points for providing details on your counter-example and why it works or details of your proof to show that the statement is true.
3. `Note:` If you choose False/True incorrectly, then you will get zero on entire problem.

! Note

If you do not have separated out proof idea and proof details for part `(b)` , you will get a zero(0) irrespective of the technical correctness of your solution..

Templates

Download LaTeX template. (latex-template.zip)

Download Microsoft Word template. (word-template.zip)

! Note

Your must explicitly list your sources and collaborators when you upload your submission to Autolab. Note that there are only five allowed sources (../../policies/hw-policy.html). If you have used a source that is not allowed, please do **not** submit your homework. If you did not consult any source or did not collaborate with anyone just say `None` .